KTH Information and Communication Technology

ID1019
Johan Montelius

Programming II (ID1019) 2018-03-13 08:00-12:00

Name: ____________________________________

Instructions

- All answers should be written in these pages, use the space allocated after each question to write down your answer.
- Answers should be written in English.
- You should hand in the whole exam.
- No additional pages should be handed in.

Grade

The exam is divided into a number of questions where some are a bit harder than others. The harder questions are marked with a star *[p*]*, and will give you points for the higher grades. The exam is thus divided into basic points and points for higher grades. First of all make sure that you pass the basic points before engaging with the higher points.

- E: 12 basic points
- D: 15 basic points
- C: 18 basic points
- B: 20 basic points and 8 higher points
- A: 20 basic points and 10 higher points

The limits could be adjusted to lower values but not raised.

Namn:________________________________ Persnr:________________________________

1 Lambda calculus [2p]

Evaluate the following lambda expressions:

- $(\lambda x \rightarrow x + 5)4$
- $(\lambda x \rightarrow (\lambda y \rightarrow x + 2 * y)3)5$
- $(\lambda x \rightarrow (x)5)(\lambda z \rightarrow z + z)$

2 Operational semantics [2p]

Given the rules for the operational semantics in the appendix, show step by step wich rules are used and evaluate the following expressions:

$$\frac{}{E\{\}(\texttt{ y = :b; y = :a; y}) \rightarrow ....}$$

$$\frac{}{E\{\}(\texttt{ \{y, y\} = \{:a, :b\}; y}) \rightarrow ...}$$

Namn:______________________________ Persnr:______________________________

3 Pattern matching [2 p]

Given the expressions below, what is the resulting environment in the cases where it succeeds?

a: [x, y | z] = [1, 2, 3]

b: [x, y | z] = [1, [2, 3]]

c: [x, y | z] = [1 | [2, 3]]

d: [x, y | z] = [1 | [2, 3] | [4]]

e: [x, y | z] = [1, 2, 3, 4]

Namn:________________________________ Persnr:________________________________

4 Recursion

a binary tree [2 p]

Implement a function, `sum/1`, that takes a binary tree and returns the sum of all values in the tree. The tree is represented as follows:

```
@type tree :: {:node, integer(), tree(), tree()} |  nil
```

tail recursion [2 p*]

The regular definition of `append/2` is not tail recursive. Implement the function `reverse/1` as a tail recursive function and use this to implement `append/2` in a tail recursive way.

Namn:________________________________ Persnr:________________________________

5 Time complexity

mirror a tree [2 p]

If we have the below definition of a function that mirrors a tree, what is the assymptoptic time complexity of teh function?

```
def mirror(nil) do nil end
def mirror({:node, left, right}) do
   {:node, mirror(right), mirror(left)}
end
```

a queue [2 p*]

Assume that we represent a queue with the help of two lists and have the below implementation of `enqueue/2` and `dequeue/1`. What is the amortized time complexity for adding and then removing an element from a queue?

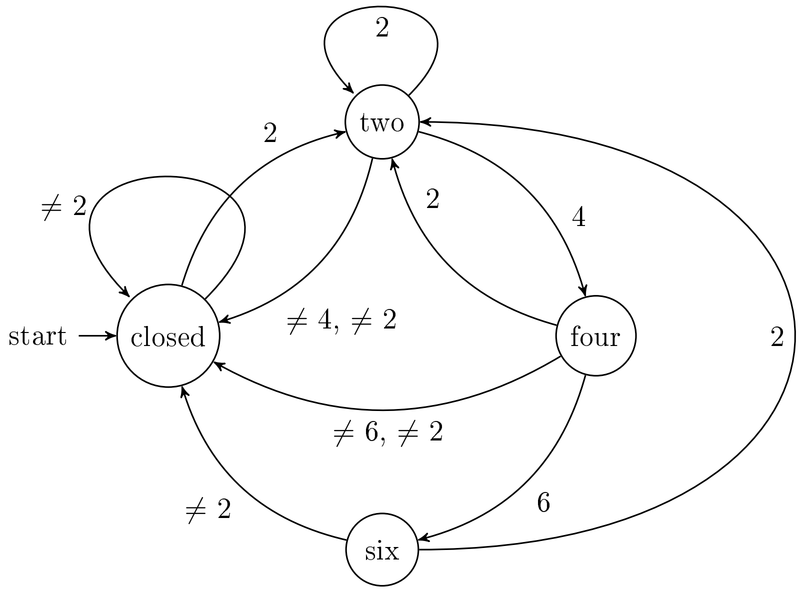
```
def enqueue({:queue, head, tail}, elem) do
  {:queue, head, [elem|tail]}
end

def dequeue({:queue, [], []}) do :fail end
def dequeue({:queue, [elem|head], tail}) do
  {:ok, elem, {:queue, head, tail}
end
def dequeue({:queue, [], tail}) do
  dequeue({:queue, reverse(tail), []})
end
```

6 Process description

TRB: two-four-six ... [2 p]

Given the below state diagram, implement a process with the specified behaviour.



tic-tac-toe [2 p]

Assume that we have the following definition of `first/1`, `second/1` and `third/1`.

```
def first(p) do
  receive do
     :tic ->
       second(p, [:tic])
     :tac ->
       second(p, [:tac])
  end
end

def second(p,all) do
  receive do
    :tic -> third(p, [:tic|all])
    :tac -> third(p, [:tac|all])
    :toe -> third(p, [:toe|all])
  end
end

def third(p, all) do
  receive do
     x ->  send(p, {:ok, [x|all]})
  end
end
```

What is the result when we evaluate the call `test/0`?

```
def test() do
   self = self()
   p = spawn(fn()-> first(self) end)
   send(p, :toe)
   send(p, :tac)
   send(p, :tic)
   receive do
     {:ok, res} -> res
   end
end
```

parallel sum [2 p*]

Implement a finction `sum/1` that takes a binary tree with numbers in the leafs, and sums all numbers of the tree in parallel.

Namn:________________________________ Persnr:________________________________

7 Programming

A heap is a tree structure where the largest element is in the root of the tree and where the left and right branch are heaps.

a heap [2 p]

Define a data structure that is suitable to represent a heap and implement a function `new/0` that returns a heap. Assume that we only should handle integers.

- `@spec new() :: heap()`

add/2 [2 p]

Implement the function `add/2` that adds an integer to a heap.

- `@spec add(heap(), integer()) :: heap()`

To keep the heap balanced you should swith the left and right branches that is, when yiu add an element to a branch you add it to teh right branch but make the result the left branch of the heap.

pop/1 [2 p]

Implement the function `pop/1` that removes the highest elemet in an heap and returns either `:fail`, if the heap is empty, or `{:ok, integer(), heap()}`

- `@spec pop(heap()) :fail | {:ok, integer(), heap()}`

swap/2 [2 p]

Implement the function `swap/2` that takes a heap and a number and returns `{:ok, integer(), heap()}` where the number is the highest number and the heap the remaining heap. The function should have the same meaning as first `add/2` a number to a heap and then `pop/1` the higest but we should do this in one fuction, not call the two functions.

- `@spec swap(heap(), integer()) {:ok, integer(), heap()}`

a generall heap [2 p]

The heap we now have will have the largest element in the root and be limited to integers (or what is comparable with `<`). Implement a function `add/3` that takes a heap, an element and a function that can be used to compare two elements. The function `add/3` should as before add an element to a heap but now used the provided function when doing the comparison.

- `@type cmp() :: (any(), any()) -> bool()`
- `@spec add(heap(), any(), cmp()) :: heap()`

Specify a type `cheap()` that holds a function for comparision and a heap. Implement a function `new/1` that takes a funktion and returns a structure of the type `cheap()`.

- `@spec new(cmp()) :: cheap()`

Implement a function `add/2` that takes a structure of type `cheap()`, that calls `add/3` with the correct arguments and returns a structure of the same form.

- `@spec add(cheap(), any()) :: cheap()`

the middle number

You should implement a process that holds a state consisting of a set of numbers. The set is initally empty but the process should then accept the following messages:

- `{:add, integer()}` : the number should be added to the set
- `{:get, pid()}` : The process should reply with either `:fail`, if the set is empty, or `{:ok, integer() }` where the integer is the middle number in the set (if an even number any one of the two middle numbers) that is also removed from the set.

To komplicate matters both operations should be done in $O(lg(n))$ time, where n is the number of elements in the set. You are not allowed to use any libraries to store the set of elements but should use the implementation of a heap in the previous questions. The process will apart from keeping track of the middle element have two heaps, one for smaller elements and one for larger.

state diagram [2*]

Start by describing a state diagram. The process should have three states: `empty` when the set is empty, `left` when we have a middle element and possibly one more element to the "left" (that are less) then we have to the "right" and, `right` when we have a middle element and possibly one more elements to the "right" than we have to the "left".

Namn:______________________________ Persnr:______________________________

no elements [2*]

Implement how we start the process and its behavior in its empty state.

Assume that we have the following functions defined in a module `Heap`.

- `@spec new(cmp()) :: cheap()`

maybe more to the left [4*]

In its "left" state the process has a middle element, a number of elements that are less (to the left) and as many or one less that are greater (to the right). Implement the behaviour of the process in its left state.

Assume that we have the following functions defined in a module `Heap`.

- `@spec add(cheap(), any()) :: cheap()`
- `@spec pop(cheap()) :: :fail | {:ok, any(), cheap()}`
- `@spec swap(cheap(), any()) :: {:ok, any(), cheap()}`

You don't have to implement the "right" state since this state will be identical to the left state apart from doing the opposit.

Namn:________________________________ Persnr:________________________________

Appendix - operational semantics

pattern matching

$$\frac{a \equiv s}{P\sigma(a, s) \to \sigma} \qquad \frac{a \not\equiv s}{P\sigma(a, s) \to \text{fail}}$$

$$\frac{v/t \notin \sigma}{P\sigma(v, s) \to \{v/s\} \cup \sigma} \qquad \frac{v/t \in \sigma \qquad t \not\equiv s}{P\sigma(v, s) \to \text{fail}}$$

$$\frac{v/s \in \sigma}{P\sigma(v, s) \to \sigma} \qquad \frac{}{P\sigma(\_, s) \to \sigma}$$

$$\frac{P\sigma(p_1, s_1) \to \sigma' \land P\sigma'(p_2, s_2) \to \theta}{P\sigma(\{p_1, p_2\}, \{s_1, s_2\}) \to \theta}$$

$$\frac{P\sigma(p_1, s_1) \to \text{fail}}{P\sigma(\{p_1, p_2\}, \{s_1, s_2\}) \to \text{fail}} \qquad \frac{P\sigma(p_1, s_1) \to \sigma' \land P\sigma'(p_2, s_2) \to \text{fail}}{P\sigma(\{p_1, p_2\}, \{s_1, s_2\}) \to \text{fail}}$$

scoping

$$\frac{\sigma' = \sigma \setminus \{v/t \quad | \quad v/t \in \sigma \quad \land \quad v \quad \text{in} \quad p\}}{S(\sigma, p) \to \sigma'}$$

expressions

$$\frac{a \equiv s}{E\sigma(a) \to s} \qquad \frac{v/s \in \sigma}{E\sigma(v) \to s} \qquad \frac{v/s \notin \sigma}{E\sigma(v) \to \perp}$$

$$\frac{E\sigma(e_1) \to s_1 \qquad E\sigma(e_2) \to s_2}{E\sigma(\{e_1, e_2\}) \to \{s_1, s_2\}} \qquad \frac{E\sigma(e_i) \to \perp}{E\sigma(\{e_1, e_2\}) \to \perp}$$

$$\frac{E\sigma(e) \to t \qquad S(\sigma, p) \to \sigma' \qquad P\sigma'(p, t) \to \theta \qquad E\theta(\text{sequence}) \to s}{E\sigma(p = e; \text{sequence}) \to s}$$

$$\frac{E\sigma(e) \to t \qquad S(\sigma, p) \to \sigma' \qquad P\sigma'(p, t) \to \text{fail}}{E\sigma(p = e; \text{sequence}) \to \perp}$$

$$\frac{E\sigma(e) \to \perp}{E\sigma(p = e; \text{sequence}) \to \perp}$$